Find out the best data-structure to represent/store the graph in memory.

An adjacency list was chosen as the optimal data structure for representing and storing the graph for both problem 1 and 2 as solution for problem 2 applies to problem 1

Problem 1: Homogenous amalgamated Star: S_n,3 (Ashna)

This adjacency list is maintained within a Python class named Graph, which also encapsulates additional graph attributes such as n, k, order, vertex labels, and edge weights. The adjacency list itself is implemented as a dictionary, where each key corresponds to a vertex and its associated value is a list of the vertex's neighbors. This implementation facilitates efficient access to the neighbors of any given vertex, which is crucial for the performance of the graph labeling algorithm utilized in the project. This approach ensures both access simplicity and storage efficiency, aligning with effective graph manipulation and data handling.

Problem 2: Homogenous amalgamated Star: S_n,m (Melisa)

(Also serves as solution to problem 1) As in Problem 1, an adjacency list within a Graph class is used to store the graph's structure. This adjacency list is implemented using a dictionary where keys represent vertices, and values are lists of adjacent vertices, facilitating efficient access to neighbors necessary for the graph labeling algorithm. Additionally, Problem 2 introduces an edge set, implemented as a Python set, named edge_weights_set. This set maintains a collection of possible edge weights that are yet to be used. The utilization of a set data structure here is pivotal for ensuring that the assignment of edge weights is unique and efficient, as sets inherently prevent duplicate entries and allow for rapid checks of membership, additions, and deletions. The approach of returning the minimum value in the set provides an O(1) retrieval much faster than incrementing a value in searching through an array of edges until the value is not in the edge_weights array.

- Adjacency list: It uses an array of linked lists or arrays to represent the graph, where each element represents a vertex and its adjacent vertices. It is efficient for sparse graphs and consumes less memory. It allows efficient traversal of adjacent vertices.

- Edge list: It represents the graph as a list of edges, where each edge contains the source and destination vertices. It is simple and memory-efficient, but it may require additional data structures for efficient traversal and edge lookup.

Devise an algorithm to assign the labels to the vertices using vertex k-labeling definition. (Main Task)

Problem 1:

a. **Algorithm Pseudocode** (vertex_k_labeling())

i. **Set label 1 for the central vertex**

ii. **Check if the total number of inner branches (internal nodes) modulo 4 equals 0, 2, or 3**

1. If true, then proceed with labeling.

iii. **Labeling internal vertices:** For each vertex i from 1 to n:

1. If i is in range from 1 to ceiling of $n / 4$ plus 1:

   a. Set label of the vertex i to $3 \times i – 2$.

2. Else, if i is in the range from ceiling of $n / 4$ plus 1 to n:

   b. Set label of the vertex i to $2 \times \text{ceiling}(n / 4) + i$.

iv. **Labeling external vertices:** Increment vertex index by 1.

v. For each i in range from 1 to ceiling of $n / 4$ plus 1:

   1. For each j in the range from 1 to 2:

      a. Set the label of the current vertex to $j + 1$.

      b. Increment the vertex index by 1.

vi. For each i in range from the ceiling of $n / 4$ plus 1 to n:

   1. For each j from 1 to 2:

a. Set label of the current vertex to \(n + i + j – 1 – 2 \times \text{ceil}(n / 4)\).

b. Increment vertex index by 1.

vii. **Return vertex labels.**

Pseudocode Image

$\phi(x_c) = 1$ **Case 1:** $n \equiv 0, 2, 3 \pmod 4$

$$\phi(x_i) = \begin{cases} 3i - 2, & \text{for } 1 \leq i \leq \lceil \frac{n}{4} \rceil + 1 \\ 2\lceil \frac{n}{4} \rceil + i, & \text{for } \lceil \frac{n}{4} \rceil + 2 \leq i \leq n \end{cases}$$

$$\phi(y_j^i) = \begin{cases} j + 1, & \text{for } 1 \leq i \leq \lceil \frac{n}{4} \rceil \text{ and } j = 1, 2 \\ n + i + j - 1 - 2\lceil \frac{n}{4} \rceil, & \text{for } \lceil \frac{n}{4} \rceil + 1 \leq i \leq n \text{ and } j = 1, 2 \end{cases}$$

Case 2: $n \equiv 1 \pmod 4$

$$\phi(x_i) = \begin{cases} 3i - 2, & \text{for } 1 \leq i \leq \lceil \frac{n}{4} \rceil \\ 2\lceil \frac{n}{4} \rceil + i - 1, & \text{for } \lceil \frac{n}{4} \rceil + 1 \leq i \leq n \end{cases}$$

$$\phi(y_j^i) = \begin{cases} j + 1, & \text{for } 1 \leq i \leq \lceil \frac{n}{4} \rceil - 1 \text{ and } j = 1, 2 \\ 2, & \text{for } i = \lceil \frac{n}{4} \rceil \text{ and } j = 1 \\ n - \lceil \frac{n}{4} \rceil + 3, & \text{for } i = \lceil \frac{n}{4} \rceil \text{ and } j = 2 \\ n + i + j - 2\lceil \frac{n}{4} \rceil, & \text{for } \lceil \frac{n}{4} \rceil + 1 \leq i \leq n \text{ and } j = 1, 2 \end{cases}$$

For more details, you can refer to the research paper: Asim, Muhammad & Hasni, Roslan & Ahmad, Ali. (2019). Edge irregular k-labeling for several classes of trees. Utilitas Mathematica. 111. 75-83.

Problem 2:

a. **Algorithm Pseudocode**

i. Initialize vertex_labels with zeros for all vertices.

ii. Set the label of the central vertex to 1.

iii.Initialize d <- k/(n-1)

iv. Initialize leaf_verts to track the numbering of leaf vertices, starting with the number of branches (n).

v. Initialize current_label to 0 for internal vertices labeling progression.

vii.vertex_labels[central] ← 1

viii. For each branch from 1 to n:

1. If branch equals 1:

   a. Set vertex_labels[branch] to 1.

   b. Calculate the edge weight between the central vertex and this branch.

   c. Assign this weight for both directions in edge_weights.

   d. Remove this weight from edge_weights_set.

2. Else:

   a. Increment current_label by d.

   b. Set vertex_labels[branch] to floor(current_label).

   c. Calculate the edge weight between the central vertex and this branch.

   d. Assign this weight for both directions in edge_weights.

   e. Remove this weight from edge_weights_set.

ix. For each branch from 1 to n:

1. For each leaf from 1 to m (exclusive)

   a. Find the minimum available weight not assigned to any edge (find_min_weight).

   b. Increment leaf_verts to get the next leaf vertex identifier.

   c. Calculate the label for this leaf vertex as the difference between leaf weight and branch's vertex label.

d. Assign this label to leaf_verts in vertex_labels.

e. Assign leaf_weight to the edge between this leaf and its branch in both directions in edge_weights.

f. Remove leaf_weight from edge_weights_set.

x. Return vertex_labels

What design strategy you will apply, also give justifications that selected strategy is most appropriate.

The design strategy used appears to be a systematic approach for assigning labels to vertices of a graph based on certain rules and conditions. It doesn't neatly fit into a single category of algorithms, but it shares characteristics with constructive algorithms. Constructive algorithms build a solution piece by piece, following a set of rules or conditions. In this case, the algorithm constructs vertex labels for a graph according to a predefined procedure based on the number of vertices and their positions within the graph. It doesn't optimize a specific objective function or make choices based on optimization criteria as seen in many optimization algorithms.

How traversing will be applied?

Traversing the graph to label the vertices is using BF or Breadth-First labeling. The internal vertices are labeled first (the ones directly connected to the center of the graph). Then finally the leaf nodes or external vertices are labeled last.

Store the labels of vertices and weights of the edges as an outcome.

Problem 1:

Edges are stored in the edge_weights data member of the Graph class, which is implemented as a dictionary. Vertex labels are similarly maintained within the vertex_labels data member of the Graph class.

Problem 2:

Graph data of labels and edge weights along with estimated time complexity will be stored in an output file named graph_output.txt once a run completes. The file will be overwritten with another run so save files as desired after each run.

Compare your results with mathematical property and tabulate the outcomes for comparison.

NOTE: Execution time includes visualization of the graph. Uncomment the call for visualization to see strictly graph construction time.

Graph Type	Execution Time (ms)	Time Complexity
S_3,2	283	24
S_3,3	303	45
S_4,4	348	144
S_7,5	452	630
S_8,6	516	1200
S_12,4	515	1200
S_4,12	518	1200
S_15,5	664	2850
S_20,7	20*	9940

Hardware resources supported until what maximum value of n, m.

Using the test_limits function, the system is specifically designed to construct graphs with n starting from 3 and incrementing by 1, while keeping m fixed at 4. This targeted approach focuses on testing the maximum n values to understand the upper limits of what the hardware can support. By incrementing only n, we can systematically assess the graph complexity that the hardware can handle without being confounded by changes in m. Although this method does not provide a comprehensive evaluation of all possible configurations, it still offers a valuable estimation of hardware limitations based primarily on the size of n. This is crucial for determining the scalability and performance thresholds specific to the given hardware environment.

Compute the Time Complexity of your algorithm T(V,E) or T(n).

compute_time_complexity function outputs the estimated time complexity to the graph_output.txt file.

a. Vertex Labeling (`vertex_k_labeling` method):

- Setting the label for the central vertex takes constant time, O(1).

- Labeling internal vertices involves a loop that iterates from 1 to `n`, performing constant time operations within each iteration. So, it has a time complexity of O(`n`).

- Labeling external vertices also involves a loop that iterates from 1 to ceil(`n`/4) + 1, performing constant time operations within each iteration. So, it has a time complexity of O(`n`).

- Overall, the time complexity of vertex labeling is O(`n`).

b. Calculating Edge Weights (`calculate_edge_weights` method):

- Iterating over the adjacency list takes O(V + E) time, where V is the number of vertices and E is the number of edges.

- For each vertex and its neighbors, calculating the edge weight involves constant time operations.

- Therefore, the time complexity of calculating edge weights is O(V + E).

c. Adding Edges to the Graph (`add_edge` method):

- Adding an edge involves appending vertices to the adjacency list, which is an O(1) time complexity operation.

- Therefore, the time complexity of adding edges depends on the number of edges added, but in this case, it's O(E).

d. Combining These Operations:

- Vertex Labeling: O(`n`)

- Calculating Edge Weights: O(V + E)

- Adding Edges: O(E)

- Since the dominant operations are vertex labeling and calculating edge weights, the overall time complexity of the algorithm would be O(`n` + V + E), which simplifies to O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Overall Time Complexity Calculation Example: Time Complexity = O(25 + 24) = O(49

Prompt

1. Suggest a suitable name.
2. Devise the formulae for calculating order and size of the graph.
3. Data-structure to store the graph.
4. Assign the labels using algorithm.
5. Store the labels of vertices and weights of the edges as an outcome.

Problem 3 (Alex)

1. A suitable name for the graph is Snowflake Star Graph.
2. **Formulas for calculating order and size of the graph:**
    a. The order of a graph is the number of vertices it has. For this graph:
    b. Order = 1 (center node) + n (branch nodes) + 2n (leaf nodes)
    c. So the formula for the order is: Order = 1 + 3n
    d. The size of a graph is the number of edges it has. For this graph:
        - There are n edges connecting the center to the branch nodes.
        - Each branch node is connected to 2 leaf nodes, so there are 2n such edges.
        - Each branch node is connected to 2 other branch nodes (forming a cycle), so there are n such edges, but since each edge connects two branch nodes, these are already accounted for and don't need to be doubled.
    e. So the formula for the size is: Size = n + 2n + n = 4n
1. **Data structure to store the graph:** The graph is stored using an adjacency list, which is suitable for sparse graphs. Each node has a list of its neighbors, and this is efficient for the graph operations required here. A dictionary structure in Python is used to maintain the adjacency list and another to maintain the edge labels.
2. **Assign the labels using an algorithm:** In the approach to graph labeling, an edge irregular k-labeling strategy was adopted where the label of an edge is defined as the maximum of the labels of the two nodes it connects. This method ensures that all edges receive unique labels, a property critical for applications such as frequency assignment in radio networks and circuit layout designs. Importantly, the inclusion of edges between branch nodes means our graph transcends a simple tree structure, necessitating a reassessment of the k value upper bound. Consequently, the upper bound is refined to k = (number of edges)×log2(number of vertices), reflecting the increased complexity and connectivity of the graph. The

vertex labeling follows a deliberate pattern, starting with the initial labeling of the central node as 1 to anchor the structure. From there, the internal branch nodes begin their labels at 11, incrementing by an alternating scheme that assigns every other odd number to ensure differentiation among them. This pattern extends to the leaf nodes, with each first leaf node labeled 1 unit higher than its parent branch node, and every second leaf node receiving a label 3 units higher. This meticulous labeling strategy ensures that each vertex has a distinct identification, facilitating the unique identification of each edge by the sum of its vertices' labels, thus optimizing the graph's overall labeling for clarity and efficiency in its intended applications.

3. **Store the labels of vertices and weights of the edges as an outcome:** The vertex labels and edge weights (labels) are stored in dictionaries. The vertex_labels dictionary maps each vertex to its label, and the edge_labels dictionary maps each edge (represented as a tuple of vertices) to its label. This is efficient for lookup and update operations and is a common way to store such information in graph algorithms.